instruction while already in the main program would lead to undesirable results when the microprocessor fetches reloads the PC with an incorrect return address.

3.4 RESET AND INTERRUPTS

Thus far, the steady-state operation of a microprocessor has been discussed in which instructions are fetched, decoded, and executed in an order determined by the PC and branch instructions. There are two special cases in which the microprocessor does not follow this regular pattern of operation. The first case is at power-up, when the microprocessor must transition from an idle state to executing instructions. This transition sequence is called *reset* and involves the microprocessor fetching its boot code from memory to begin the programmed software sequence. Reset is triggered by asserting a particular logic level onto a microprocessor pin and can occur either at power-up or at any arbitrary time when it is desired to restart, or reboot, the microprocessor from a known initial state. Some microprocessors have special instructions that can actually trigger a soft reset.

The question arises of how the microprocessor determines which instruction to execute first when it has just been reset. To solve this problem, each microprocessor has a *reset vector* that points it to a fixed, predetermined memory address where the programmer must locate the first instruction of the boot sequence. The reset vector is specified by the microprocessor's designer. Some microprocessors locate the reset vector at the beginning of memory and some place it toward the end of the address space. Sometimes the main body of the program will be located in another portion of memory, and the first instruction at the reset vector will contain a branch instruction to jump to the desired location.

The second case in which the microprocessor does not follow the normal instruction sequence is during normal operation when an event occurs and the programmer wishes the microprocessor to pause what it is currently doing and handle the event with a special software routine. Such an event is called an *interrupt*. A common application for an interrupt is the implementation of a periodic, timed operation such as monitoring the temperature of a room. Because the room temperature does not change often, the microprocessor can handle other tasks during normal operation. A timer can be set to expire every few seconds, causing an interrupt event. When the interrupt triggers, the microprocessor can read the room temperature, take any appropriate action (e.g., turn on a ventilation fan), and then resume its normal operation.

An interrupt can be triggered by asserting a special-purpose microprocessor interrupt signal. Interrupt events can also be triggered from within a microprocessor via special instructions. When an interrupt occurs, the microprocessor saves its state by pushing the PC and other registers onto the stack, and then the PC is loaded with an *interrupt vector* that points to an *interrupt service routine* (ISR) in memory. In this way, the interrupt process is similar to a branch-to-subroutine. However, the interrupt may be triggered by an external hardware event instead of by software. Like reset, each interrupt pin on the microprocessor has an interrupt vector associated with it. The programmer knows that an ISR is to be located at a specific memory location to service a particular interrupt. When the ISR has completed, a *return-from-interrupt* instruction is executed that restores the microprocessor's prior state by popping it from the stack. Control is then returned to the routine that was interrupted and normal execution proceeds.

As the interrupt mechanism executes, the program that gets interrupted does not necessarily have any knowledge of the event. Because the state of the microprocessor is saved and then restored during the return-from-interrupt, the main routine has no concept that somewhere along the way its execution was paused for an arbitrary period. The programmer may choose to make such knowledge available by sharing information between the ISR and other routines, but this is left to individual software implementations.

Multiple interrupt sources are common in microprocessors. Depending on the complexity of the microprocessor, there may be one, two, ten, or dozens of separate interrupt sources, each with its own vector. Conflicts in which multiple interrupt sources are activated at the same time are handled by assigning priorities to each interrupt. Interrupt priorities may be predetermined by the designer of the microprocessor or programmed by software. In a microprocessor with multiple interrupt priorities, once a higher-priority interrupt has taken control and its ISR is executing, lower-priority interrupts will remain pending until the current higher-priority ISR issues a return-from-interrupt.

Interrupts can usually be turned off, or *masked*, by writing to a control register within the microprocessor. Masking an interrupt is useful, because an interrupt should not be triggered before the program has had a chance to set up the ISR or otherwise get ready to handle the interrupt condition. If the program is not yet ready and the microprocessor takes an interrupt by jumping to the interrupt vector, the microprocessor will crash by executing invalid instructions.

Masking is also useful when performing certain time-critical operations. A task may be programmed into an ISR that must complete within 10 μs. Under normal circumstances, the task is easily accomplished in this period of time. However, if a competing interrupt is triggered during the time-critical ISR, there may be no guarantee of meeting the 10-μs requirements. One solution to this problem is to mask subsequent interrupts when the time-critical interrupt is triggered and then unmask interrupts when the ISR has completed. If an interrupt arrives while masked, the microprocessor will remember the interrupt request and trigger the interrupt when it is unmasked.

Certain microprocessors have one or more interrupts that are classified as nonmaskable. This means that the interrupt cannot be disabled. Therefore, the hardware design of the computer must ensure that such an interrupt is not activated unless the software is able to respond to it. Nonmaskable interrupts are generally used for low-level error recovery or debugging purposes where it must be guaranteed that the interrupt will be taken regardless of what the microprocessor is doing at the time. Nonmaskable ISRs are sometimes implemented in nonvolatile memory to ensure that they are always ready for execution.

3.5 IMPLEMENTATION OF AN EIGHT-BIT COMPUTER

Having discussed some of the basic principles of microprocessor architecture and operation, we can examine how a microprocessor fits into a system to form a computer. Microprocessors need external memory in which to store their programs and the data upon which they operate. In this context, external memory is viewed from a logical perspective. That is, the memory is always external to the core microprocessor element. Some processor chips on the market actually contain a certain quantity of memory within them, but, logically speaking, this memory is still external to the actual microprocessor core.

In the general sense, a computer requires a quantity of nonvolatile memory, or ROM, in which to store the boot code that will be executed on reset. The ROM may contain all or some of the microprocessor's full set of software. A small embedded computer, such as the one in a microwave oven, contains all its software in ROM. A desktop computer contains very little of its software in ROM. A computer also requires a quantity of volatile memory, or RAM, that can be used to store data associated with the various tasks running on the computer. RAM is where the microprocessor's stack is located. Additionally, RAM can be used to hold software that is loaded from an external source.

For purposes of discussion, consider the basic eight-bit computer shown in Fig. 3.7 with a small quantity of memory and a serial port with which to communicate with the outside world. Eight kilobytes of ROM is sufficient to store boot code and software, including a serial communications program. Eight kilobytes of RAM is sufficient to hold data associated with the ROM software, and it also enables loading additional software not already included in the ROM. The control signals in this